Uniform, Versatile and Efficient Dense and Sparse Multi-Dimensional Arrays

Wolfgang Hoschek*
CERN IT Division - European Organization for Nuclear Research
1211 Geneva 23, Switzerland
wolfgang.hoschek@cern.ch

Abstract

We describe a Java toolkit for rectangular dense and sparse multi-dimensional arrays. Such a toolkit should be usable, efficient and allow for the definition of many algorithms from disparate domains. Balancing these competing requirements is challenging. We address this issue by providing a few non-overlapping general-purpose concepts that a user can combine in a straightforward, uniform and efficient manner to generate complex behaviour. The most important aspect of this approach is that a single polymorphic interface provides uniform access to any matrix or view thereof. Every matrix and view class implements the same interface with exactly the same semantics, no more, no less. Therefore, everything that can be done with a dense matrix can - without code change - also be done with a sparse matrix, and vice-versa. Everything that can be done with a normal matrix (dense or sparse) can - without code change - also be done with any (abitrarily nested) matrix view, and vice-versa. We describe non-delegating true matrix views for sub-ranging, striding, transposition, slicing, index flipping, cell selection, as well as sorting, permuting and partitioning. We also describe a space and time efficient sparse representation based on hashing. We then compare the performance with three other toolkits, including C/C++. Throughputs up to 500 MB/s and 100 Mflops/s are demonstrated in the dense case, and normalized up to 24 GB/s and 3 Gflops/s in the sparse case. In every benchmark, our toolkit is among the leading two. Performance is better than C/C++. Our overall conclusion is that rich and complex behaviour cannot only be provided with minimal usability impact, but also with high efficiency.

1 Introduction

Rectangular multi-dimensional arrays, sometimes also called Data Cubes, are rectangular grids with each cell of the grid holding a single value. They can be characterized by immutable rank (the number of dimensions or axes), immutable shape (the number of slots in each dimension) and immutable value type (typically integer, floating point or an arbitrary object). Cells are accessed via non-negative integer indexes based on zero (C/C++, Java) or one (Fortran). Thus, 1-d, 2-d and 3-d arrays correspond to the mathematical concepts of vector, matrix and discrete 3-d space, respectively. We will subsequently refer to them as matrices, in order to avoid confusion with Java's classic arrays of arrays, Ninja Arrays and various other classes carrying array names. Example 2-d and 3-d matrices are depicted in Figure 1.

Dense and sparse rectangular multi-dimensional arrays are of critical importance for scientific and technical computing, as they are the most fundamental abstraction used in this domain. They are, for example, essential for Linear Algebra, Statistics, Data Mining, Online Analytic Processing (OLAP), Computational Fluid Dynamics and Climate Simulation, to name just a few. For example, many physics problems can be mapped to matrix problems such as Linear and nonlinear systems of equations as well as linear differential equations. Data analysis in High Energy Physics uses multi-dimensional arrays in the process of discovering meaningful new correlations, patterns and trends by scanning over data, using statistical and mathematical techniques as well as pattern recognition technologies. Financial OLAP applications analyze the typically large and sparse data cube built on sales by product by year by shop [Sho97]. Time complexity of operations on multi-dimensional arrays is typically demanding - $O(N^2)$ and $O(N^3)$ is the normal rather than the pathologic case. As a result, performance is seen as an integral part of usability.

*Also affiliated with Dep. of Information Systems, Institute of Applied Computer Science, University Linz, Austria

Java - like most modern programming languages - is a lean language pushing flexibility and expressive power (i.e. rich functionality) out of the language into user space. Thus, it provides only rudimentary support for the manipulation of rectangular multi-dimensional arrays, i.e. retrieving and modifying a cell from a dense array. More comprehensive support is to be provided at the library level.

Such a library should meet several requirements. It should be easy to use both for novices and experts, highly efficient, portable, flexible and expressive enough to allow for the formulation of a multitude of algorithms from many disparate domains. It should be possible to treat sparse matrices no different than their dense counterparts so that a user can choose a matrix implementation purely based on density, not based on supported functionality. A library should also be open in a way that encourages involvement by the community at large, leading to improvements and extensions over time.

Some of these requirements are competing. Their sensible balance poses a challenge. Easy to use toolkits tend to be limited in functionality. Toolkits can be designed for simplicity or for scalability in terms of performance and complexity. The former approach initially shows low perceived complexity, but its learning curve explodes with the number of supported operations. The latter approach initially shows higher perceived complexity, but a more consistent learning curve even in the presence of a large number of supported operations. Portable toolkits cannot easily exploit system specific parameters and thus tend to be less efficent than toolkits fully optimized for a given architecture. On the other hand side, portable Java toolkits make disappear altogether many annoying problems related to incompatible or unsupported compilers, linkers and architectures. Flexible toolkits are not necessarily efficient (generality and indirection can restrict performance) and easy to use (explosion of concepts, policies and modules). Tightly focussed toolkits can be too specialized to foster strong productivity and reuse. Toolkits with expressive power can provoke statements like "I don't need 95% of this functionality and have trouble to find the 5% I do need." The practical relevance of a highly efficient but barely usable toolkit is similar to that of a toolkit fulfilling all requirements except that performance is unacceptable. Design is - and will remain - a challenge.

In this paper, we present a Java toolkit[1] that strives to balance these factors in sensible ways. The toolkit navigates the space spanned by {*usability, performance, expressive power*}. We address this issue by providing a few non-overlapping general-purpose concepts that a user can combine in a straightforward, uniform and efficient manner to generate complex behaviour. The most important aspect of this approach is that a single polymorphic interface provides uniform access to any matrix or view thereof. Every matrix and view class implements the same interface with exactly the same semantics, no more, no less. Therefore, everything that can be done with a dense matrix can - without code change - also be done with a sparse matrix, and vice-versa. Everything that can be done with a normal matrix (dense or sparse) can - without code change - also be done with any (abitrarily nested) matrix view, and vice-versa. The major contributions of this paper are five fold.

- We introduce a uniform architecture providing expressive power with minimal negative impact on usability
- We introduce space and time efficient hash based sparse matrices
- We introduce true selection views as well as an efficient non-delegating implementation
- We provide a production quality implementation of all concepts
- We compare the properties and performance of several related toolkits

2 Functionality

The main properties of our Colt toolkit are i) Complete general-purpose functionality, ii) Rich functionality for special-purpose domains, iii) Efficient dense and sparse data structures and multi-purpose operations, iv) Expressive power with a minimum of concepts, maintaining a high degree of usability, v) Uniform polymorphic interfaces to dense and sparse matrices, vi) True matrix views with the same uniform interface as normal dense and sparse matrices, vii) Efficient non-delegating implementation of views, viii) Open Source.

The Colt toolkit provides dense and sparse matrix data structures in 1-d, 2-d and 3-d, with variants holding either `double` floating point or `Object` cells. Matrices are accessed via zero-based non-negative integer indexes. They are equipped with a set of multi-purpose operations for assignment, copying, element-by-element transformation,

[1] This work is carried out in the broader context of the Colt distribution, providing production quality Open Source Libraries for High Performance Scientific and Technical Computing in Java, and available at `nicewww.cern.ch/~hoschek/colt/index.htm`.

comparison, aggregation, relaxation, print formatting and conversion from/to normal Java arrays. To allow users to efficiently formulate complex operations with simple terms, matrix data structures are also equipped with operations generating true matrix views for sub-ranging, striding, transposition, slicing, index flipping, cell selection, as well as sorting, permuting and partitioning. Views behave exactly like - and can be used in place of - normal matrices. All matrices and views of a given rank and value type implement a uniform interface without sub-setting or super-setting, and with exactly the same semantics. Implementations are encapsulated and completely hidden to users. They can arbitrarily be combined and nested. The powerful and lean concept of true views will later be discussed in more detail.

Since a great many algorithms from many different domains can be defined on matrices, a single interface cannot accomodate all of them without seriously compromising usability and maintainability. Thus, for scalability in terms of complexity, special purpose algorithms are pushed into external packages and classes. These include special element-by-element transformations, matrix decompositions from Linear Algebra, customizable print formatting, 1-d and 2-d Statistics and other algorithms.

Operations carry transactional properties. That is, they check preconditions early and throw appropriate exceptions. Factory classes with a uniform interface allow for the convenient construction and decomposition of dense and sparse matrices. While discussion and examples concentrate on 2-d and 1-d matrices holding `double` floating point values, matrices in 3-d and/or holding `Object` values support analogous functionality. Portability is adressed by using pure Java. The toolkit runs with a single code base and shared Java library on any Virtual Machine compatible with JDK 1.2.0 or higher, for example under Linux, Solaris and NT.

3 True views

With views any matrix can be presented to the user and/or functions in various ways. They follow a spreadsheet like interaction pattern: Users select a region of cells, then issue commands on the region. Views are virtual transformations. Constructing a view does not involve copying or modifying the data. Views merely alter the way data is shown. Carl and Jane can look at the same flower in many different ways, but it always remains the same flower. If Carl steps on top of the flower (modifies data), Jane will certainly note. True views carry reference semantics, adhere to the same interface as normal matrices and can arbitrarily be combined and nested. To avoid interface bloat they should be orthogonal in the sense that their functionality does not overlap. In our context, true views do not maintain semantic constraints in the sense that, for example, a sort view automatically remains sorted under update.

Currently true views in all dimensionalities are supported for sub-ranging, striding, transposition, slicing, index flipping, cell selection as well as sorting, permuting and partitioning. The 2–d and 3-d case is visualized by Figure 1. More detailed explanation is contained in Table 1. Views for sorting, permuting and partitioning are provided for convenience only; they are reduced to selection views.

4 Usability and Expressive Power

In this section, we argue that rich and complex behaviour can be supported with simple means. Matrix toolkits tend to provide expressive power (i.e. rich and complex functionality) at the cost of usability, often exposing to the user large amounts of classes and operations differing only in small details. For example, toolkits without true views define different get/set methods for normal matrices and sub-ranged strided matrices:
`A.get(i,j)` versus `A.get(i,j,rowStride,columnStride,fromRow,fromColumn)`
Similar operations are defined for selected matrices, etc. In order to multiply a subranged matrix with another subranged matrix, a special multiply operation with additional parameters defining the subranges needs to be called - and remembered. Such a function could look like
`A.multRange(B,C,fromRowA,toRowA,fromColA,toColA,fromRowB,fromColB,toColB,fromRowC,fromColC)`. An additional operation may be available to multiply a strided subranged matrix with another matrix - with an even longer argument list. There may be no operation allowing to multiply a permuted matrix with a strided transposed subranged matrix, because this would require additional implementation effort and the argument list would become unmanageably long. Further, distinction of dense and sparse matrices effectively doubles the number of classes and interfaces.

As another example, each element-by-element transformation (e.g. `abs,mult,div,plus,minus`) may come in one, two or three versions, either modifying the target matrix (`A.assignAbs()`), an additionally supplied result matrix (`A.abs(B)`, or constructing a new result matrix (`B=A.abs()`). Even a small number of concepts translates to

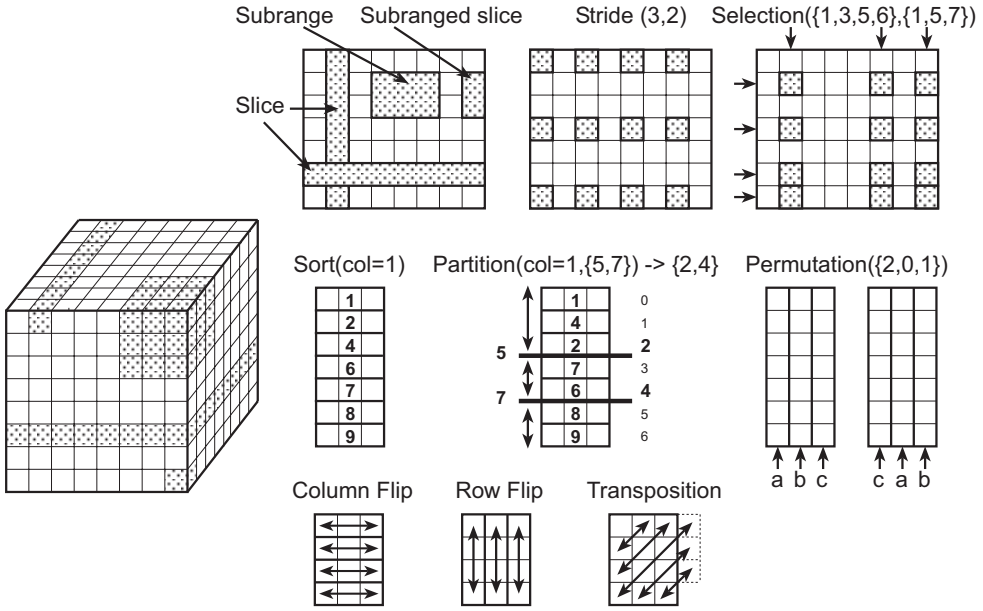

Figure 1: 2-d and 3-d matrix views

View name	*Description*
sub-range	Shows contigous subregion of cells; specified by coordinate, height and width
stride	Shows every u-th, v-th cell
transposition	Shows exchanged axes (dimensions); e.g. turns rows into columns and columns into rows
slice	Shows lower dimensional matrix by fixing one dimension at a given index
flipping	Shows indexes mirrored along a given dimension; e.g. turns sorted ascending into descending
selection	Shows the given subset of rows and/or columns in the specified order Shows rows matching an arbitrary condition function
sorting	Shows rows sorted by a given column or by an arbitrary comparison function
permutation	Shows rows and/or columns rearranged according to the given order
partition	Partitions the domain of a column into non-overlapping sub-domains (intervals). Shows rows rearranged such that all rows falling into an interval are moved next to each other. Returns indexes of interval delimiters. Frequently used prior to efficient aggregation or histogramming.

View name	*Definition*	*Interface*	*Example*
sub-range	V[i,j]=A[i+r,j+c]	viewPart(r,c,h,w)	V=A.viewPart(1,3,2,3)
stride	V[i,j]=A[i*u,j*v]	viewStrides(u,v)	V=A.viewStrides(3,2)
transposition	V[i,j]=A[j,i]	viewDice()	V=A.viewDice()
slice	V[i]=A[j,i] V[i]=A[i,j]	viewRow(j) viewColumn(j)	V=A.viewRow(5) V=A.viewColumn(1)
flipping	V[i,j]=A[A.rows-1-i,j] V[i,j]=A[i,A.columns-1-j]	viewRowFlip() viewColumnFlip()	V=A.viewRowFlip() V=A.viewColumnFlip()
selection	V[i,j]=A[rows[i],cols[j]] rows matching condition	viewSelection(rows,cols) viewSelection(condition)	V=A.viewSelection({1,3,5,6}, {1,5,7})
sorting	rows sorted by column rows sorted by function	viewSorted(j) viewSorted(comparison)	V=A.viewSorted(1)

Table 1: Definition of 2-d matrix views

a combinatorical explosion of interfaces which are very hard to use, to implement and to maintain over time. For example, despite a small number of implemented concepts, the Ninja toolkit [MMG99] defines no less than 160 methods on a 2-d matrix. Other toolkits exceed 1000 methods. Even if a needed operation exists, it remains a challenge to find and remember it.

Striving to provide expressive power with minimal usability cost, we take a different approach, avoiding combinatorical explosion of interfaces. We provide a rich set of functionality with a minimum of concepts (classes, methods, notations, policies). That is, by providing a few entirely independent (orthogonal) concepts that users can combine in a straightforward and uniform manner, yielding combinatorically complex functionality. The cornerstone of this approach is the concept of

- A single polymorphic interface providing uniform access to any matrix or view thereof
- Every matrix and view class implements the same interface with exactly the same semantics, no more, no less

This has the following consequences

- Everything that can be done with a dense matrix can - without code change - also be done with a sparse matrix, and vice-versa.
- Everything that can be done with a normal matrix (dense or sparse) can - without code change - also be done with any (abitrarily nested) matrix view, and vice-versa.
- Matrices of any kind can be nested and combined in any conceivable way and subsequently manipulated as if they were a plain normal matrix.
- View classes need not be exposed to the user. A view is not generated by explicitly instantiating a class, but by calling a method defined on the matrix to be viewed. Thus, views carry little conceptual complexity.
- Combinatorical explosion of interfaces is avoided. No unnecessary complexity is exposed to the user. Interfaces remain lean. No code changes are required.

The following code snippet shows basic usage of the toolkit.

```
DoubleMatrix2D A,B,C;                    // declare as 2-d with value type 'double'
DoubleMatrix1D D,E;                      // declare as 1-d with value type 'double'
A = new DenseDoubleMatrix2D(7,8);        // construct dense 7 x 8 matrix
B = new SparseDoubleMatrix2D(7,8);       // dense and sparse matrices have same interface 'DoubleMatrix2D'
C = A.viewPart(1,3,2,3).viewSorted(0);   // construct nested subrange view sorted by first column
D = C.viewColumn(0);                     // views have same uniform interface as 'normal' matrix
E = B.viewColumn(0);
DoubleMatrix3D F = new DenseDoubleMatrix3D(8,8,8); // construct dense 8 x 8 x 8 matrix
DoubleMatrix2D G = F.viewRow(5);                   // view a 2-d slice
DoubleMatrix1D H = G.viewColumn(0).viewStrides(2); // view every second cell of first column of 2-d slice
```

A matrix cell is retrieved and modified with `A.get(row,column)` and `A.set(row,column,value)`, respectively, no matter whether dense, sparse, subranged, strided, permuted, sorted, sliced, etc, or a combination thereof. A new matrix holding copied cell values is generated via `B=A.copy()`. Matrix cell values are copied into another matrix with `A.assign(B)`. A sparse matrix can be multiplied with a dense transposed matrix by using the same method that multiplies any two arbitrary matrices (`A.zMult(B,C)` versus `A.viewDice().zMult(B,C)`) A transposed, strided, subranged view can not only be sorted by a given column, but it can be sorted in exactly the same way a normal matrix is sorted (`S=A.viewSorted(0)` versus `S=A.viewDice().viewStrides(1,2).viewPart(0,0,4,4).viewSorted(0)`).

Any operations defined on rows (e.g. sorting) can be made to work on columns, simply by using a transposition view instead of the original matrix. Element-by-element transformations come in one version only, modifying the matrix in-place (`A.assign(Functions.abs)`). Transformations modifying supplied result matrixes or constructing new result matrices are not necessary because they can be emulated with minimal performance impact by combining the transformation, assign and copy method. For example, as in `B.assign(A).assign(Functions.abs)` or `B=A.copy().assign(Functions.abs)`, respectively. To summarize, we argue that rich and complex behaviour can be supported with simple means.

5 Implementation

In this section we argue that rich and complex behaviour can not only be provided with minimal usability impact, but also with high efficiency. We show how the architecture described thus far can be implemented.

All Matrices address linear storage space, regardless of dimensionality and type. Linear storage space is addressed C-like in row-major and slice-major, respectively. Dense matrices encapsulate a contigous one-dimensional primitive Java array. Sparse matrices use hashing on a contigous one-dimensional primitive Java array based on automatically resizing open adressing with double hashing. Our hashmap implementation is a separately reusable component. It is carefully tuned to deliver good throughput with little memory consumption and efficient use of the memory hierarchy.

Memory consumption. Memory consumption of dense matrices is a function of the number of cells: $bytes = rows \times columns \times 8$. Example: A_{10^6} or $A_{1000\times1000}$ or $A_{100\times100\times100} \rightarrow mem = 8MB$. Sparse matrices are very general in the sense that they are not bound to any specific pattern of non-zero cells. They equally well accomodate arbitrary and special sparse patterns such as diagonal, band diagonal, tridiagonal, block, column banded and random matrices because memory consumption is not a function of a non-zero pattern, but a function of the number of non-zero cells: $\frac{nonZeros\times13}{maxLoadFactor} \leq bytes \leq \frac{nonZeros\times13}{minLoadFactor}$. Example: A_{10^9} or $A_{32000\times32000}$ or $A_{1000\times1000\times1000}$, 10^6 non-zeros, default $minLoadFactor = 0.25, maxLoadFactor = 0.5 \rightarrow 25MB \leq mem \leq 50MB$. This property eliminates much of the need for ad-hoc special purpose representations.

Get/Set performance. Getting/setting a cell value takes *guaranteed* constant time for dense matrices and all their views, while it takes *expected* constant time for sparse matrices and all their views. More specifically, on sparse hash matrices, these operations can degenerate to time linear in the number of non-zero cells. Such is the nature of hashing: Average case behaviour is excellent, worst case behaviour is poor. Theory establishes that such pathologic cases are exceedingly improbable and can artificially be generated only with entirely non-intuitive sequences of `set`, which moreover change dynamically. In accordance with theory [Knu73], very few, if any, hash collisions are observed in practice. In our experiments sparse get/set performance is only 3-4 times slower than in the dense case!

Million ops/sec	*dense*	*sparse*	*speedup*
get	12	3	4
set	7	2.2	3.2

Although no experimental evidence is currently available, complexity and constant factor analysis suggests that with comparable space efficiency, get/set on sparse hash matrices is one or more orders of magnitude faster than Coordinate Storage and Column or Row-Compressed Storage. For practical purposes this means that for many applications specialized sparse data structures may turn out to be unnecessary.

Since views are implemented without delegation, get/set on strided, sub-ranged, transposed, flipped and sliced views (and arbitrary combinations thereof) shows exactly the same constant factors as on non-views, no matter how deeply nested they are. Permutation, partition and sort views are convenience sugar and reduced to selection views. Selection views are also implemented without delegation. Get/set functionality on selection views requires no additional function call, but does require one additional primitive array lookup per dimension, leading to slightly increased constant factors. Again, the level of nesting is irrelevant.

View construction. Views are constructed in guaranteed constant time, except for selection views and sort views. Selection views take time linear in the number of indexes, sort views take time $O(N \times logN)$ on average. Note the implication of views with reference semantics: Cells are not physically moved or copied, leading to excellent performance in particular in the presence of large matrices.

Non-delegating views. Views can be implemented with or without delegation. Delegating views use coordinate transforming wrapper objects, non-delegating views adjust cell addressing parameters to the same effect but without any indirection. Note that interface and semantics are identical in either case. Due too indirection, delegating views are subject to serious performance degradation. First, a single get/set triggers a chain of function calls cascading through all levels of nested views. Such chains can be long and hard (if not impossible) to inline for compilers. Second, kernel loops cannot manually reduce cell adressing overhead to a mere integer increment, as is consistently done in all dimensionalities for dense views and all their view combinations, except for selection views. The result is that delegating views can be 1-2 orders of magnitude slower than non-delegating views. We believe that in the absence of highly optimizing compilers and/or sophisticated C++ expression templates (as used in Blitz++ and Pooma-2), a matrix toolkit based on delegating views would be unacceptable for high performance applications.

Uniform cell adressing formula. Linear cell addressing can be implemented in a number of increasingly general

ways:

$$addr = row \times columns + column \tag{1}$$

$$addr = offset + row \times rowStride + column \times columnStride \tag{2}$$

$$addr = rowZero + row \times rowStride + columnZero + column \times columnStride \tag{3}$$

Note that Java's classic arrays of arrays and (1) cannot support non-delegating views, that (2) supports non-delegating subrange, transposition, slice and flip views, but does not easily lend itself to selection views. (3) is prepared for non-delegating selection views (and hence non-delegating sort, permutation and partition views), using the following formula (4)

$$addr = offset + rowOffsets[rowZero + row \times rowStride] + columnOffsets[columnZero + column \times columnStride] \tag{4}$$

Let us now examine key implications of formula (3) and (4). The code for cell addressing is identical for all matrices of a given dimensionality. Cell addressing in any dimensionality is derived by straightforward extension. The difference between dense and sparse implementation of get/set is kept minimal: `array1D[addr]` vs. `hashmap.get(addr)` and `array1D[addr]=value` vs. `hashmap.put(addr,value)`, respectively. Arithmetic overhead is often hidden by pipelining. Further, dense toolkit kernel loops can and do avoid overhead by reducing cell adressing to a single integer increment. View construction can be implemented easily: The code adjusting cell addressing parameters is identical for all matrix classes and needs to be implemented only once in an abstract base class, as next shown for some views.

transpose	$rows' = columns; columns' = rows; rowStride' = columnStride; columnStride' = rowStride;$
stride	$rows' = \frac{rows}{u}; columns' = \frac{columns}{v}; rowStride' = rowStride \times u; columnStride = columnStride \times v$

To summarize, in our context, arrays of arrays are both inflexible and inefficient. The synergetic effects of a uniform linear cell addressing formula translate to easy implementation as well as consistent and good performance. Thus, all matrices use the same strided 32-bit integer arithmetic, as given in formula (3), except for selection views using formula (4).

Kernel loops. Despite the fact that Java Just-in-time and adaptive compilers have strongly improved over the last two years, it is still considered risky to exclusively rely on the compiler for compute intensive tasks. Therefore, multi-purpose operations are optimized by hand. All operations directly defined on matrix classes are optimized kernel loops. They are the building blocks for efficient user applications. Note that optimizations are not exposed at the interface level. Employed optimizations include i) Blocking for memory latency hiding ii) Function call and cell address overhead elimination, iii) Loop unrolling, iv) Passing safe regions to the compiler, guaranteed to complete without throwing any exception, v) Transparent sparsity detection, vi) Transparent small/large matrix specialization.

To summarize, we argue that rich and complex behaviour can not only be provided with minimal usability impact, but also with high efficiency.

6 Experimental Results

In this section the performance of several important core operations is evaluated. Results of Colt 1.0 Beta4.1 are compared with the Ninja 0.3, Jama and CLHEP 1.4.0 toolkits. Colt, Ninja and Jama are written in Java, CLHEP in C/C++. CLHEP serves as the base line reference, because it is a highly optimized implementation. It is, for example, used in on-line reconstruction of high troughput data acquisition systems at CERN.

Environment and methodology. Experiments are conducted on a 2 x PentiumIII@600MHz, 32 KB L1, 512 KB L2, 512 MB running IBMJDK1.1.8 on Red Hat Linux 6.0, Kernel 2.2.5-22smp. The IBM port is chosen because it consistently equals or outperforms BlackdownJDK1.2.2RC3 and SunInpriseJDK1.2.2RC1. Each operation is timed varying implementation type (`DenseDoubleMatrix2D`, `SparseDoubleMatrix2D`), matrix size (all matrices are square with "size" rows and "size" columns, plotted on the x-axis) and density (the fraction of cells in non-zero state, randomly assigned). Measurements are given in Mops/sec (million operations/sec) and Mflops/sec (million floating point operations/sec). $A_{i,j} = B_{k,l}$ counts as 1 operation whereas $sum = sum + A_{i,j} \times B_{k,l}$ counts as 2 floating point operations. For sparse data types, Mops and Mflops are expressed in relation to the dense base line implementation: If an operation on a dense datatype with 100% density executes at 10 Mflops/sec but takes twice as long to complete on a sparse matrix, the sparse matrix is said to have a performance of $10/2 = 5$ Mflops. This metric is useful to compare user perceived performance. All machines are empty. Garbage collection is not explicitly invoked. Each operation is repeated for at least 2 seconds. The mean of all repetitions is reported. The java `-ms30m -mx120m` options are used. CLHEP is compiled with egcs 1.1.2 and the `-O3` optimization option. Only one of the two available CPUs is used.

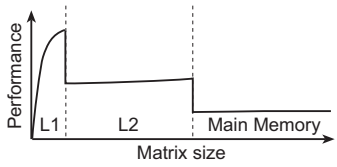

Figure 2: Memory hierarchy latency dominating performance

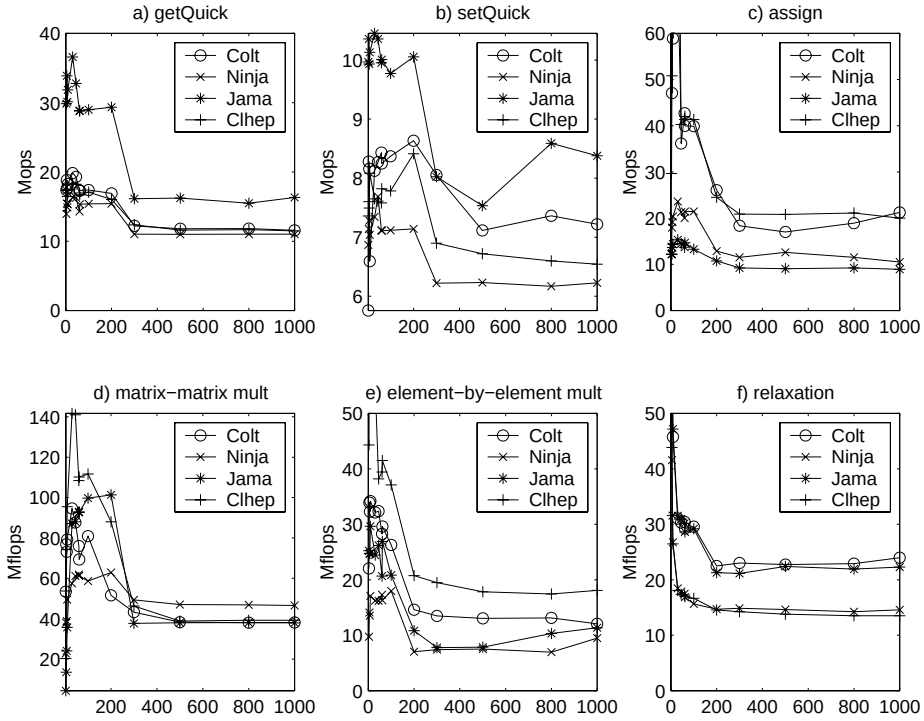

Figure 3: Dense 2-d matrix performance

6.1 Dense Matrix

Figure 3 depicts the performance of several core operations on dense matrix data structures, as executed by the Colt, Ninja, Jama and CLHEP toolkit.

Overall observations. The performance of all toolkits is similar. The clock cycle of CPUs is increasing much faster than memory latency is decreasing. As a result, the shape of all curves is dominated by latency of the memory hierarchy and follows the associated well known pattern: With tiny matrix size, operations start out slow (due too function call and initialization overhead) but rapidly accelerate to peak performance. Upon overflow of Level-1 cache, they drop to a plateau where they stay until Level-2 cache overflows and performance drops to the last plateau, now running on main memory. This behaviour is idealized in Figure 2. The largest observed speedup of Colt over CLHEP is 1.8 and seen for relaxation (f).

Get/set (a,b). 12-22 vs. 7-8 Mops. These operations sum up or initialize all cell values, respectively. They appear to be successfully inlined by the compiler. Otherwise they could not reach such good performance, as compared to optimized assignment. Interestingly, Jama is fastest on get/set, even though its primitive 2-d Java arrays are considered inefficient [MMG99]. Since get/set are inlined anyway, the reason does not seem to be the fact that Jama breaks encapsulation and allows to access internal matrix state without function call. Rather, the difference is a result of the inability of current compilers to optimize cell addressing arithmetic. Row aliasing of primitive Java 2-d arrays incurs little cell arithmetic. The advantage of primitive 2-d Java arrays fades or is inverted for "whole

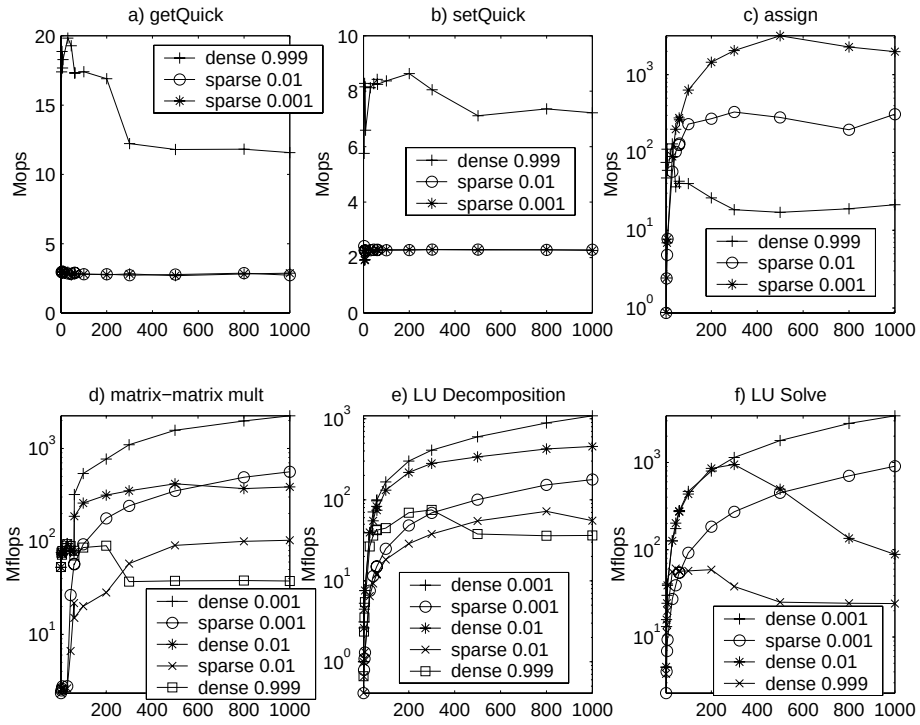

Figure 4: Sparse 2-d matrix performance

array" operations, because all toolkits manually reduce cell addressing to integer increments (pointer increments for CLHEP), while Jama's primitive 2-d arrays can only do so in special cases.

Assignment (c). 160-1000 MB/sec. Assignment copies the contents of one matrix into another one. It represents the upper limit on memory bandwidth and thus can serve as frame of reference for other operations. CLHEP and Colt as leaders show very good and nearly identical performance ($20 \times 8 = 160MB/sec$). Without PentiumIII-specific prefetch assembler instructions no significant improvement is to be expected.

Matrix-matrix mult (d). 40-100 Mflops. The Ninja toolkit performs slightly better than Colt, probably due too a better blocking algorithm. The reason remains unclear, as no source code is made available.

Relaxation (f). 25-45 Mflops. The five-point stencil $B_{i,j} = f(A_{i-1.j}, A_{i+1,j}, A_{i,j}, A_{i,j-1}, A_{i,j+1})$ with $f = \alpha \times A_{i,j} + \beta \times (A_{i-1.j} + A_{i+1,j} + A_{i,j-1} + A_{i,j+1})$ is used. Colt performs best, with speedup expected to be even more pronounced for nine-point relaxation. The reason is that CLHEP, Ninja and Jama do not directly support stencil operations. Thus, CLHEP and Ninja users need to use get/set and cannot optimize cell addressing. Jama users can manually eliminate function calls and optimize cell addressing via row aliasing because of the already mentioned access to internal state, leading to performance almost equal to Colt.

To summarize, our toolkit compares favourably despite of its general-purpose nature and because of careful implementation.

6.2 Sparse Matrix

Since the other considered toolkits do not support sparse matrices, Figure 4 compares dense against sparse Colt matrix classes. All matrices use default $minLoadFactor = 0.2, maxLoadFactor = 0.5$. As already stated, performance is expressed in relation to the dense base line implementation. Note that the last four plots are logarithmic on the y-axis. *dense* and *sparse* indicate usage of `DenseDoubleMatrix2D` and `SparseDoubleMatrix2D`, respectively. 0.01 indicates that 1% of all cells have non-zero state, chosen randomly from the uniform distribution.

Get/set (a,b). 3 vs 2.2 Mops. Get/set on sparse matrices is only 3-4 times slower than on dense matrices. Sparse matrices with very large density reported similar performance and are therefore omitted. Performance is not dominated by memory latency but by algorithmic overhead. We believe that these results are excellent.

Assignment (c). 0-3000 Mops. Sparse assignment is specialized; it only copies non-zero cells. This leads to performance 1-2 orders of magnitude better than the dense implementation.

Matrix-matrix mult (d). 0-3100 Mflops. In general, performance increases with decreasing density. In the best case, sparsity detection turns the $O(N^3)$ algorithm into $O(N^2)$. If enough memory is available it is faster by a constant factor to have a dense rather than a sparse data structure. The speedup of *dense 0.001* over *sparse 0.001* is constant 4.5 - the result of different get/set performance and manual cell address overhead elimination. The matrix size threshold for transparent switching to sparsity detection is well chosen, because nearly invisible even if detection is executed in vain, as is the case for dense matrices.

LU (e,f). 0-3200 Mflops. Similar characteristics are observed for the LU Decomposition and equation solving with LU. However, intermediate state with higher density can lead to performance degradation, as can be seen for the sparse data structure with somewhat large density. The speedup of *dense 0.001* over *sparse 0.001* is constant 6, for the same reasons mentioned for the matrix-matrix multiply.

7 Related Work

The Ninja toolkit [MMG99] provides multi-dimensional arrays in 1-3 dimensions and supports many value types. Their design is less general and extendible than ours. It does not foresee subclassing, sparse data, selection, sorting, permutation and partitioning views. The Ninja interface is easier to use than most other toolkits but still very redundant. They only provide limited linear algebraic and special purpose functionality. Ninja is used for research into optimizing native compilers. Thus, the encouraging performance published in [MMG99],[AGMM99],[MMS98] is achieved by using proprietary technologies not generally available and with varying degree of JDK compatibility. They advocate hard-wiring the semantics of their toolkit and an optimized implementation into the Virtual Machine.

Special purpose Java toolkits for dense Linear Algebra have been designed as wrappers of existing C and Fortran code [BG97], cross-compiled from Fortran to Java [DDS98] or redesigned from scratch like Jama [HMW+98]. None of them supports views or general-purpose functionality. Jama provides high quality algorithms for Linear Algebra on dense 2-d matrices. The fact that Jama does not support multi-dimensional array functionality and that Ninja does not support Linear Algebra seems unfortunate. Thus, our linear algebraic algorithms are to a large degree a one-to-one mapping of the Jama code base to our multi-dimensional array classes. The C/C++ toolkit CLHEP [CLH96] has similar properties as Jama.

Efficient C++ toolkits for multi-dimensional arrays include Blitz++ [Vel98] and Pooma-2 [KCC+98]. They provide some views not available in our work, and vice-versa. No sparse representations are available. They heavily use template expressions and operator overloading to allow for high performance, powerful generic types and notations more resembling mathematics than C++. These language features are considered double-ended swords - very powerful but tricky and generally poorly understood.

Sparse Libraries in C++ [DLPR94] provide special-purpose functionality on established representations such as Coordinate Storage, Column and Row-compressed storage, but no hashing. MOLAP databases such as Arbor's Essbase [Inc96] implement multi-dimensional arrays with hash tables designed and optimized for secondary storage, but no high performance API.

8 Conclusions

We describe a Java toolkit for rectangular dense and sparse multi-dimensional arrays. Such a toolkit should be usable, efficient and allow for the definition of many algorithms from disparate domains. Balancing these competing requirements is challenging. We address this issue by providing a few non-overlapping general-purpose concepts that a user can combine in a straightforward, uniform and efficient manner to generate complex behaviour. The most important aspect of this approach is that a single polymorphic interface provides uniform access to any matrix or view thereof. Every matrix and view class implements the same interface with exactly the same semantics, no more, no less. Therefore, everything that can be done with a dense matrix can - without code change - also be done with a sparse matrix, and vice-versa. Everything that can be done with a normal matrix (dense or sparse) can - without code change - also be done with any (abitrarily nested) matrix view, and vice-versa. We describe non-delegating true matrix views for sub-ranging, striding, transposition, slicing, index flipping, cell selection, as well as sorting, permuting and partitioning. We also describe a space and time efficient sparse representation based on hashing. We then compare the performance with three other toolkits, including C/C++. Throughputs up to 500 MB/s and 100 Mflops/s are

demonstrated in the dense case, and normalized up to 24 GB/s and 3 Gflops/s in the sparse case. In every benchmark, our toolkit is among the leading two. Performance is better than C/C++. Our overall conclusion is that rich and complex behaviour cannot only be provided with minimal usability impact, but also with high efficiency.

References

[AGMM99] P. Artigas, M. Gupta, S. Midkiff, and J. Moreira. High performance computing in java: Language and compiler issues. In *Proceedings of the 12'th Workshop on Language and Compilers for Parallel Computers*, 1999.

[BG97] A. Bik and D. Gannon. A note on native level 1 blas in java. Technical Report TR483, Department of Computer Science, Indiana University, 1997.

[CLH96] Clhep. 1996. Available at http://wwwinfo.cern.ch/asd/lhc++/clhep.

[DDS98] D. Doolin, J. Dongarra, and K. Seymour. Jlapack-compiling lapack fortran to java. Technical Report TR390, Department of Computer Science, University of Tennessee, Knoxville, 1998.

[DLPR94] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. A sparse matrix library in c++ for high performance linear algebra. In *Proc. of the Object Oriented Numerics Conference*, 1994.

[HMW+98] J. Hicklin, C. Moler, P. Webb, R. Boisvert, B. Miller, R. Pozo, and K. Remington. Jama. 1998. Available at http://math.nist.gov/javanumerics/jama.

[Inc96] Arbor Software Inc. Multidimensional analysis: Converting corporate data into strategic information. white paper. 1996. Available at http://www.arborsoft.com/papers/multiTOC.html.

[KCC+98] S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. Humphrey, J. Reynders, S. Smith, and T. Williams. Array design and expression evaluation in pooma ii. In *Lecture Notes in Computer Science, 1505, p. 231*, 1998.

[Knu73] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching.* Addison-Wesley, 1973.

[MMG99] J. Moreira, S. Midkiff, and M. Gupta. A standard java package for technical computing. In *Proc. of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[MMS98] S. Midkiff, J. Moreira, and M. Snir. Optimizing array reference checking in java programs. In *IBM Systems Journal, Aug. 1998, Vol. 37, No. 3, pp. 409-453*, 1998.

[Sho97] Arie Shoshani. Olap and statistical databases: Similarities and differences. In *Proc. of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* ACM Press, May 12-14 1997.

[Vel98] Todd L. Veldhuizen. Arrays in blitz++. In *Proc. of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.